Chapter 18: Paging: Introduction

**Paging** is to chop up space into fixed-sized pieces. Instead of splitting up a process’s address space into some number of variable-sized logical segments (code, stack, heap), we divide it into fixed-sized units, each of which called **page**. we view physical memory as an array of fixed-sized slots called **page frames**, each of these frames can contain a single virtual-memory page.

**18.1 A simple example and overview**

**Table

Description automatically generated**

In the above example, the 64-bute chunk of address space is divided into four 16-byte pages. Real addresses are much bigger.

Physical memory also consists of a number of fixed-sized slots, and pages of the virtual address space have been placed at different location throughout the physical memory. OS also uses some of the memory for itself.

A picture containing graphical user interface

Description automatically generated

The most important improvement will be **flexibility**: with a fully developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space.

Another advantage is **simplicity** of free-space management. The OS just needs to keep a free list of all free pages.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a **per-process** (the OS have to manage different one for different process) data structure known as a **page table**. The purpose of this table is to store address translations for each of the virtual pages of the address space and let us know the location of each page in physical memory.



To translate the virtual address, we split it in two components: the virtual page number and the offset within the page. Because the virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address.

Diagram

Description automatically generated with medium confidence

Since the page size is 16 bytes in a 64-byte address space, we will have 4 pages and we will need the first 2 bits (Va5 and Va4) to select one of the four pages. The remaining bits are offset.

For example, if we load the virtual address 21, its binary value is

Table, box and whisker chart

Description automatically generated

Therefore, the page number is 1 and the offset is 5 (stay the same in the physical page). In the page table (figure 18.2), **the physical frame number** (**PFN** or **physical page number** or **PPN**) is 7 because it is in the 7th page. Thus, we must translate this virtual address by replacing the VPN with PFN:

**Diagram

Description automatically generated**

**18.2 Where Are Page Tables Stored?**

For example, if we have a 32-bit address space with 4KB pages, meaning that we have 20-bit VPN and 12-bit offset.

A 20-bit VPN meaning that we have 220 translations to be managed by the OS. Assume that we need 4 bytes per **page table entry (PTE)** to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table. This is pretty large.

**18.3 What’s Actually in The Page Table?**

The page table is just a data structure that is sued to map virtual addresses to physical addresses. Therefore, any data structure would work. The simplest form is called **linear page table**, which is just an array.

As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is valid. For example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process.

Graphical user interface, text

Description automatically generated

We might have **protection bits** indicating whether the page could be read from, written to, or executed from.

A **present bit** indicates whether this page is in physical memory or on disk.

A **dirty bit** is also common indicating whether the page has been modified since it was brough into memory.

A **reference bit (accessed bit)** is sometimes used to track whether a page has been accessed and is useful in determining which pages are popular and thus should be kept in memory.

Text

Description automatically generated

The above picture contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S) which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the page frame number (PFN) itself.

**18.4 Paging: Also Too Slow**

With page tables in memory, we already know that they might be too big. They can slow things down too. For every memory reference, paging requires performing one extra memory reference in order to first fetch the translation from the page tables. It is a lot of work, and extra memory are costly, causing the process to slow down by a factor of two or more.

Text

Description automatically generated

**18.5 A Memory Trace**

A picture containing diagram

Description automatically generated

The assembly code for the above program is

Text, letter

Description automatically generated

The memory trace is depicted as follows:

Chart

Description automatically generated

The bottom graph will remain the same as the same instructions will be repeated for 1000 times. The middle graph will increase in term of memory address because the register eax will be increased over each iteration. The top graph will depend on the translation from VPN to PFN, but in general, it first fetches into the VPN and the updates the PFN. The remaining three fetches will just update VPN variables.